Étude des performances du Distributed Spanning Tree : un Overlay Network pour la Recherche de Services

Alexandru Dobrila, Sylvain Dahan, Jean-Marc Nicod, Laurent Philippe

RenPar'18 / SympA'2008 / CFSE'6
Fribourg, Suisse, du 11 au 13 février 2008

Étude des performances du *Distributed Spanning Tree* un *Overlay Network* pour la Recherche de Services

Alexandru Dobrila, Sylvain Dahan, Jean-Marc Nicod et Laurent Philippe*
LIFC - 16, Route de Gray - 25 030 Besançon cedex

Résumé

Les arbres et les graphes aléatoires sont deux topologies qui sont habituellement utilisées pour construire des réseaux de recouvrement et implanter des mécanismes de découverte à grande échelle. Nous proposons une autre topologie d'interconnexion des nœuds, le *Distributed Spanning Tree*, qui permet l'implantation d'algorithmes de traversée d'arbres tout en évitant les goulets d'étranglement inhérents à ceux-ci et, de ce fait, offre une exécution plus efficace des algorithmes de recherche en terme du nombre de messages et de répartition de la charge. Cet article présente les résultats de simulations des performances du DST. Ces simulations nous ont permis de conclure que le DST offre de bonnes performances pour les recherches, meilleures que celles des réseaux à base de graphes et meilleures que celles des arbres. Nous étudions également le comportement de la structure face à l'ajout ou à la suppression de nœuds pour montrer qu'elle est adaptée aussi aux environnements dynamiques.

1. Introduction

Les intergiciels de grille permettent le partage de ressources qui peuvent être du matériel, des données ou des applications. Avec l'accroissement de la taille des grilles, la recherche de ressources adaptées aux besoins d'un utilisateur devient une problématique en soi. Pour y répondre, les systèmes pair-à-pair proposent des mécanismes de recherche extensibles [10].
Nous pouvons classer ces mécanismes en deux catégories [11]. La première catégorie est constituée de mécanismes basés sur des répertoires (*directory*). Dans ces mécanismes, chaque ressource doit s'enregistrer en spécifiant son nom et sa localisation pour être visible des autres. Ensuite, pour retrouver cette ressource, les clients donnent le nom de la ressource au répertoire. Ce schéma est mis en œuvre par les *Distributed Hash Tables (DHT)* utilisées dans les réseaux pair-à-pair. Par contre, bien qu'elles permettent une recherche rapide et exhaustive, les DHT ne sont pas adaptées à la recherche de ressources car les requêtes sont basées sur les identifiants exacts des ressources. Elles n'ont ainsi qu'une expressivité limitée. C'est pourquoi certaines structures utilisent des plages de valeurs comme SkipGraph [1]. La seconde catégorie est composée par les mécanismes basés sur les algorithmes d'inondation ou *flooding*. Dans ce cas, les informations sur les ressources sont conservées sur chacune des machines qui participe au réseau pair-à-pair. Ces machines sont interconnectées pour former un graphe de communication, également appelé réseau de recouvrement ou *overlay network*. Lorsque l'un des pairs recherche une information, il demande à ses voisins, les nœuds adjacents du graphe. Si ceux-ci, à leur tour, ne possèdent pas l'information, ils font suivre la requête à leurs propres voisins, et ainsi de suite jusqu'à ce que l'information soit trouvée, ou jusqu'à ce qu'un nombre maximal de sauts, appelé *Time To Leave* ou TTL, ait été fait. Pour limiter le nombre de voisins contactés à chaque saut, nous pouvons utiliser un algorithme de marche aléatoire [13].
Chaque mécanisme de recherche possède ses propres avantages ou inconvénients et le choix de l'un d'entre eux repose sur les besoins spécifiques d'une application ou sur les caractéristiques d'un contexte. Par exemple, l'intergiciel de grille DIET [2] utilise un mécanisme de flooding [5] afin de trouver les serveurs disponibles pour exécuter des requêtes de méta-computing. La recherche des serveurs prend en compte deux critères : la disponibilité de l'application et une évaluation de performances, basée sur la localisation des données utilisées, le temps CPU et la charge actuelle du serveur. Comme ces informations sont dynamiques, il n'est pas possible de les mémoriser dans un répertoire car il serait trop coûteux de les maintenir à jour, ce qui justifie le recours à un algorithme de flooding.
Les performances des algorithmes de flooding dépendent de plusieurs facteurs, dont la topologie du graphe de communication. Pour ces algorithmes, la structure d'arbre est intéressante car elle permet une complexité en terme du nombre de messages optimale puisqu'elle dépend directement du nombre de nœuds dans le réseau. Cependant, quand la charge augmente, les arbres génèrent des goulets d'étranglement au niveau des nœuds proches de la racine. De ce fait, les algorithmes de flooding sont, la plupart du temps, exécutés

* Cette recherche a été effectuée dans le cadre du projet INRIA GRAAL.

sur des graphes pseudo-aléatoires[2], bien que leur complexité en nombre de messages soit supérieure. La complexité de l'algorithme dépend du nombre de liens et non du nombre de nœuds, ce qui la place au dessus des arbres puisque le nombre de liens d'un graphe est généralement supérieur à celui des nœuds.
Pour améliorer les performances des algorithmes de flooding nous proposons une nouvelle topologie pour représenter un *overlay network* appelée DST pour *Distributed Spanning Tree* [4, 6] dans laquelle chaque nœud est la racine de son propre arbre de recouvrement des nœuds du réseau. Cela permet d'éviter les goulets d'étranglement d'une topologie en arbre et de réduire le nombre de messages observés avec des graphes. Nous mettons en évidence, dans cet article, les avantages de cette topologie face aux arbres et aux graphes pseudo-aléatoires grâce à une étude comparative des performances obtenues par des recherches dans différents cas de figures. Les résultats sont obtenus par simulations. Nous étudions également le comportement du DST dans un contexte plus dynamique où les nœuds peuvent apparaître ou disparaître plus fréquemment.
La simulation a déjà été utilisée dans de nombreux cas pour étudier les performances d'algorithmes de flooding. Par exemple, G. Fletcher and H. Sheth [7] évaluent les efficacités des recherches effectuées respectivement avec une DHT, un graphe aléatoire et un graphe Pandurangan. L. Qin et Al. [12] ont comparé sur les performances et la charge supportée par différentes topologies de graphes aléatoires et différentes distributions de ressources. En marge de ces simulations, des travaux comme ceux de R. Gaeta et Al. [8] proposent des modèles analytiques pour les algorithmes de recherche sur des graphes aléatoires.
L'article est organisé comme suit. Nous rappelons d'abord ce qu'est la topologie d'un DST, puis nous décrivons au paragraphe 4 les algorithmes qui peuvent être utilisés pour parcourir un DST. Nous donnons les algorithmes de construction/suppression des nœuds d'un DST au paragraphe 3. Nos comparaisons de performances étant réalisées par simulation, nous en décrivons les enjeux et le contexte au paragraphe 5. Enfin, les résultats de ces simulations sont donnés et analysés au paragraphe 6. Nous finissons cet article par une conclusion sur les possibilités d'utilisation des DST dans divers contextes.

2. La structure des DST

La structure du DST a été conçue pour palier aux problèmes de diffusion posés par les arbres et les graphes pseudo-aléatoires dans les réseaux de recouvrement à large échelle. L'idée de base est la suivante : *"d'une part les arbres limitent le nombre de messages mais génèrent des goulets d'étranglement et, d'autre part, les graphes génèrent trop de messages mais distribuent la charge entre les pairs"*. Le principe recherché est de distribuer les arbres sur le réseau de manière à limiter le nombre de messages tout en évitant les goulets d'étranglement. Le DST est ainsi un arbre virtuel qui recouvre l'ensemble des machines et pour lequel les nœuds sont distribués pour éviter toute contention, d'où son nom le Distributed Spanning Tree. Pour ce faire, il est composé de nœuds, qui regroupent et structurent les machines en *overlay*, et composent un arbre.
Le DST possède des propriétés comparables aux B-tree [9] : il possède plusieurs niveaux, toutes les feuilles sont au même niveau et chaque nœud qui n'est pas une feuille a un nombre limité de fils. La racine possède au moins deux fils, si elle n'est pas une feuille. Le niveau 0 est composé par les feuilles de l'arbre. Les nœuds de niveau 1 sont des graphes complets de feuilles. La figure 1 présente 3 nœuds de niveau 1 et 8 feuilles. Les niveaux supérieurs sont ensuite construits en réalisant un graphe complet de leurs fils de niveau directement inférieur. La figure 2 est un exemple d'un nœud de niveau 2.

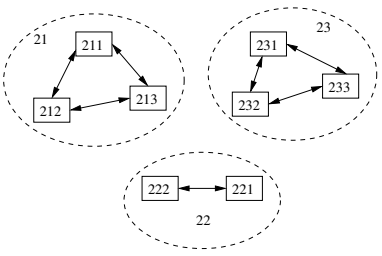

FIG. 1 – Niveau 1 d'un DST

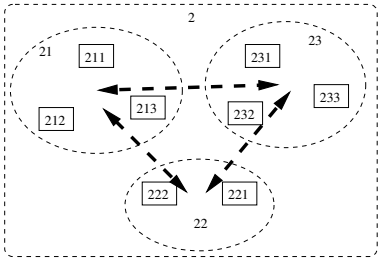

FIG. 2 – Niveau 2 d'un DST

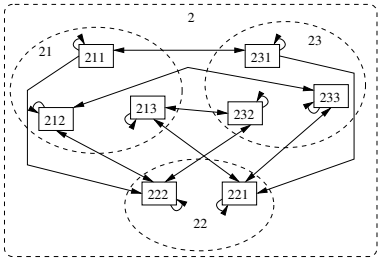

FIG. 3 – Liens entre nœuds

Dans un environnement à large échelle, il n'est pas raisonnable de supposer que cette structuration peut être imposée par un membre du réseau ou une entité de régulation. Il est donc nécessaire de définir une

[2] un graphe pseudo-aléatoire est construit en ajoutant/retirant des nœuds et des liens sans dépendance entre ces ajouts/suppressions.

mise en œuvre distribuée pour ne pas simplement déplacer les goulets d'étranglement et pour limiter les risques liés aux pannes. Par définition, un nœud d'un DST, qui n'est pas une feuille, est alors distribué entre ses descendants et donc distribué entre les ordinateurs qui participent au nœud. De ce fait, un nœud n'a d'existence qu'à travers les machines qui s'y insèrent, c'est une entité logique.
Cette structuration qui permet de regrouper logiquement les machines doit également décrire les interconnexions supportant les algorithmes utilisés sur les *overlay network*. Dans un premier temps, il est facile de communiquer de manière descendante d'un père vers un fils puisque le premier constitue un réseau complet du second. Cependant, l'entité père est logique, comme dit précédemment. Il est donc nécessaire de mettre en place des liens réels entre ordinateurs du DST. Pour définir un nœud de niveau n, avec $n > 2$, nous connectons chacun des ordinateurs de ce nœud avec au moins un des ordinateurs de chacun de ses fils, comme cela est montré sur la figure 3. Ainsi, chaque machine du nœud peut facilement envoyer un message à chaque fils de ce nœud grâce aux liens qu'il possède vers ces fils. Cette même figure illustre qu'un nœud physique appartient à chaque niveau de hiérarchie. Par exemple, l'ordinateur 222 appartient également aux nœuds logiques 22 et 2.
Sur la base de ces règles de construction, chaque nœud génère ses propres tables de routage. La table de routage d'une machine possède donc autant d'entrées que de niveaux dans le DST et chaque entrée contient un lien vers un ordinateur pour chacun des nœuds frères dans ce niveau. De plus, chaque ordinateur possède un représentant pour chaque nœud logique et il est son propre représentant dans les nœuds dont il fait partie. Par exemple, sur la figure 3, si le nœud 2 veut diffuser un message à ses descendants, il va envoyer, d'un point de vue logique, un message aux nœuds 21, 22 et 23. Cette diffusion peut être initiée par n'importe quel ordinateur. Si elle est concrètement initialisée sur l'ordinateur 213, il va envoyer à son représentant du nœud 21, donc lui-même, du nœud 22 qui est, pour lui, la machine 221 et du nœud 23 qui est la machine 232. Ensuite, ces représentants des nœuds 22 et 23 diffuseront à leur tour le message à leurs enfants en utilisant leur graphe d'interconnexion de niveau 1.
Cette explication illustre le principe simple et fondateur de la topologie d'un DST. Cette topologie est d'une part logique, les ordinateurs appartiennent à tous les niveaux d'une hiérarchie, et d'autre part physique avec la définition pour chaque ordinateur d'une table de routage individuelle, de taille logarithmique, permettant le recouvrement de toutes les machines du réseau. Une construction qui distribue les arbres de recouvrement garantit la répartition de la charge entre les différentes machines du réseau et évite ainsi les goulets d'étranglement. Une description plus complète de la structure du DST est donnée dans [6, 3].
Dans la partie suivante nous présentons comment sont gérés les nœuds du DST.

3. Gestion du DST

La gestion du DST est incrémentale : nous ajoutons ou retirons des ordinateurs aux nœuds en fonction de l'arrivée ou du départ des participants. Pour faire partie du DST, un nœud isolé doit connaître un autre nœud faisant déjà partie de la structure. Afin de maintenir une structure permettant de conserver de bonnes propriétés quant à la recherche, nous introduisons la notion de bornes sur la taille d'un nœud du DST. Nous définissons une borne inférieure en dessous de laquelle un nœud doit fusionner avec un autre nœud de même niveau et une borne supérieure au delà de laquelle un nœud doit se scinder en deux nœuds distincts.

3.1. Ajout de nœuds

Soit n un ordinateur à insérer, n connaît le nœud *contact* qui appartient déjà au DST. Afin d'accepter un nouveau membre, il faut qu'il y ait de la place dans son groupe. Si le groupe de premier niveau est plein, il faut le diviser, ce qui ajoute un élément dans le groupe de niveau supérieur. Si ce groupe, de niveau 2, est lui aussi de taille maximale, il faut aussi le diviser, et ainsi de suite, récursivement jusqu'au plus haut niveau.
À ce niveau, nous savons combien de groupes vont devoir être divisés. Nous divisons alors les groupes les uns après les autres par ordre de niveau décroissant, si cela est nécessaire. Par contre, il se peut qu'aucune division ne soit nécessaire.
Si la racine est divisée, un nouveau niveau est créé car le DST n'a qu'une racine. Une fois les niveaux ajoutés, nous informons les nœuds concernés par les scissions. La même opération est répétée pour tous les nœuds à scinder.
Lorsque toutes les scissions sont faites, la dernière étape consiste, pour *contact*, à informer son groupe de l'arrivée du nouveau nœud (n). Les nœuds du groupe ainsi que le nœud n peuvent alors mettre à jour leur table de routage. Cette dernière étape peut être la seule si aucune scission est nécessaire.

3.2. Algorithme de suppression de nœuds

Nous souhaitons ici, supprimer le nœud *self*. Une suppression peut entraîner la nécessité de fusionner des nœuds devenus trop petits. L'algorithme 1 passe en revue tous les cas de suppressions possibles.

Algorithme 1 : Algorithme de suppression d'un nœud du DST

Fonctions :
effaceFrere(niveau, nom) : efface dans la liste des frères d'un nœud, le nœud *nom* au niveau *niveau*.
obtenirNvlConnection(niveau) : permet d'obtenir une nouvelle connection dans le DST au niveau *niveau*.
ajouteFrereA(niveau, indice, nvo) : ajoute dans la liste des frères au niveau *niveau* le nœud *nvo* à l'indice *indice*.
ajoutePrd(niveau, nvo) : ajoute dans la liste des prédécesseurs au niveau *niveau* le nœud *nvo*.

```
1  début
2      (fusion, aFusionner) ← self.nettoieNiveauZero() ;
3      pour i de 0 à nbNiveaux − 1 faire
4          pour chaque Jdest de self.pred[i] faire
5              ind ← Jdest.effaceFrere(i, self.nom) ;
6              nvlConnection ← self.obtenirNvlConnection(i) ;
7              Jdest.ajouteFrereA(i, ind, nvlConnection) ;
8              nvlConnection.ajoutePred(i, Jdest) ;
9          pour chaque Kdest de self.frere[i] autre que self faire
10             Kdest.effacePred(i, self.name) ;
11     si fusion ≠ 0 alors
12         pour chaque elt de aFusionner faire
13             elt.demandeFusion(0) ;
14         si dernier élt de aFusionner n'a pas assez de frères au niveau 1 alors
15             si nbNoeudsRestant > MAXFRERE alors
16                 dernier.demandeHauteFusion(1, dernier.frere[0]);
17             sinon
18                 dernier.diffusionSuppressionNiveau();
19 fin
```

Nous traitons le cas où une suppression est programmée. Nous considérons que le nœud qui quitte le DST a le temps de prévenir de son départ. Nous commençons donc par prévenir les nœuds du niveau 1 (l. 2). Cette étape est primordiale. Elle permet de déterminer la nécessité de fusionner ou non des groupes. Ainsi fusion est la variable qui indique s'il faut ou non fusionner et aFusionner est une liste contenant les nœuds concernés. Pour chaque niveau de DST (l. 3), les tables de routage des prédécesseurs (resp. des frères) sont mises à jour (l. 4-8 et resp. l. 9-10).
Lors de la suppression de *self*, s'il n'y a plus assez d'éléments dans le groupe, il faut fusionner les membres du groupe avec un autre groupe (l. 11). Tous les éléments de la liste aFusionner vont faire une demande auprès d'un voisin de niveau 2 (l. 12-13). À ce niveau, une fusion équivaut à la suppression d'un groupe de niveau 1. Il faut alors tester s'il reste assez d'éléments au niveau 1 pour conserver la structure (l. 14). Deux choix sont possibles : soit il reste assez de frères pour conserver le niveau (l. 15), nous effectuons alors une fusion de plus haut niveau (l. 16) ; soit, nous manquons d'éléments et devons supprimer un niveau entier (l. 18). Dans le dernier cas, il faut contacter tous les nœuds du DST pour les tenir informés.

3.3. Performances

Pour évaluer le coût de gestion du DST nous avons simulé des ajouts successifs puis des suppressions.

3.3.1. Coût des ajouts

Pour simuler les ajouts, nous créons un DST avec 1 million de nœuds avec des bornes inférieure et supérieure sur la taille des groupes respectivement de 4 et 8. Les ajouts sont réalisés successivement les uns après les autres sans notion de fréquence. La première série de simulations donne le coût d'ajouts de nœuds dans

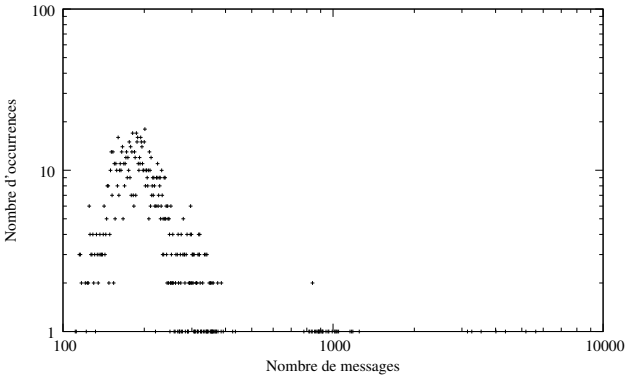

FIG. 4 – Nombre de messages avec 4 étages.

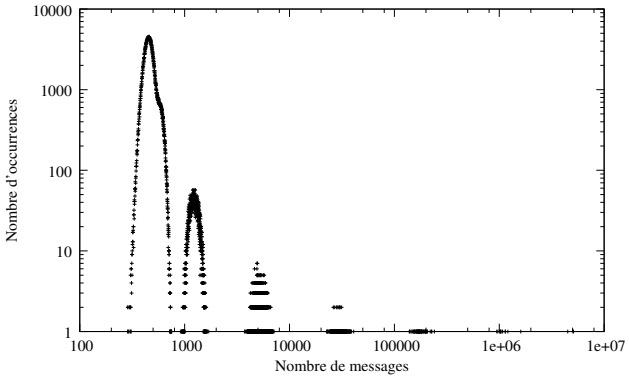

FIG. 5 – Nombre de messages avec 8 étages.

le DST. Les figures 4 et 5 montrent la répartition du nombre de messages par rapport au nombre de niveaux. Les échelles sont logarithmiques. L'abscisse correspond au nombre de messages nécessaires à l'ajout d'un sommet. L'ordonnée correspond au nombre de sommets qui ont conduit au nombre de messages indiqué en abscisse pour leur insertion. Pour lire ces courbes, il est utile de se rappeler que chaque insertion d'un nœud produit un certain nombre de messages et qu'il est possible que deux insertions produisent le même nombre de messages.
Nous remarquons l'apparition de pics lorsque le nombre d'étages est supérieur à 4. Chaque pic correspond aux nombres de messages nécessaires à la scission d'un nœud d'un étage différent. Plus le nœud est haut plus le nombre de sommets concernés par la scission est important. Ceci engendre un nombre important de messages.
Ces figures montrent que le nombre de messages nécessaires à la scission d'un nœud croît de manière exponentielle avec la hauteur du nœud. Cette figure montre également que le nombre de scissions d'un nœud diminue de façon logarithmique en fonction de sa hauteur dans le DST.
Ainsi, il y a beaucoup de scissions de nœuds des premiers étages qui sont peu coûteuses en nombre de messages et peu de scissions de nœuds des derniers étages qui sont très coûteuses en nombre de messages. Dans la simulation montrée ici, il y a un rapport de dix entre les deux.

3.3.2. Coût des suppressions

Pour évaluer le coût des suppressions, nous avons réalisé plusieurs simulations qui visent à mesurer le coût de différents facteurs intervenant dans la suppression.

Impact des zones de suppressions

Nous testons le comportement du DST face à des suppressions ciblées. Pour ces simulations, nous utilisons un DST de 50 nœuds auquel nous supprimons 30 nœuds. Les bornes inférieure et supérieure pour la taille des nœuds valent respectivement 2 et 4.
Dans un premier temps, nous effectuons des suppressions aléatoires dans le DST. Ce résultat nous sert de témoin dans la suite. Dans un second temps, nous ciblons les suppressions dans une zone précise du DST, c'est à dire que les suppressions vont avoir lieu toujours au niveau du même groupe, forçant ainsi les réorganisations. Nous lançons 20 simulations, pour chaque zone, et retenons le nombre total de messages nécessaires pour effectuer les 30 suppressions. Les deux courbes obtenues sont représentées à la figure 6.
En moyenne, la suppression aléatoire a besoin de 1374 messages pour supprimer 30 nœuds. La suppression ciblée en a besoin de 1555. Ce résultat s'explique par la probabilité plus élevée, dans le cadre d'une suppression ciblée, de déclencher une fusion de 2 groupes devenus trop petits. Une bonne gestion algorithmique de la réorganisation est donc primordiale. Nous nous penchons d'avantages sur les coûts de celle-ci dans le paragraphe suivant.

Coût de la réorganisation

Nous utilisons ici un DST avec les plus petites bornes possibles, à savoir 2 et 4. Initialement, le DST possède 90 nœuds. Cela nous donne une structure organisée en cinq niveaux. À partir de ce DST, nous supprimons 88 nœuds afin d'étudier toutes les étapes de fusion passant d'un DST à 5 niveau à un DST à 1 niveau. Les résultats ont été effectués sur trois simulations et sont représentés à la figure 7.
Nous remarquons sur cette figure que les séries 2 et 3 connaissaient un pic du nombre de messages juste

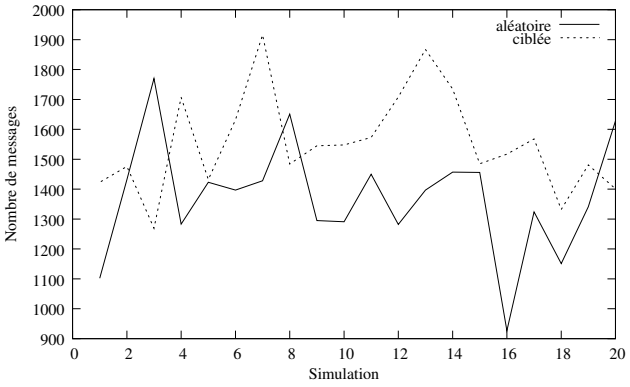

FIG. 6 – Étude des zones de suppressions.

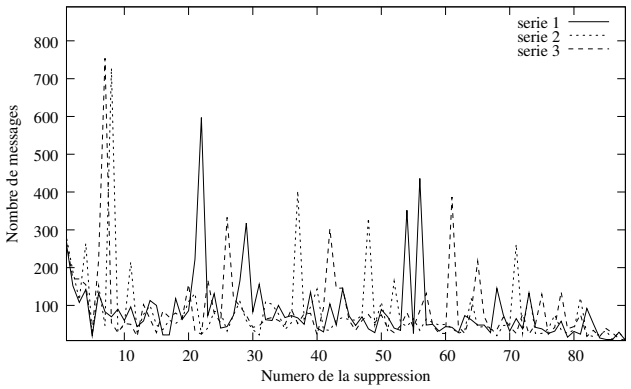

FIG. 7 – Coût de la réorganisation.

avant la dixième suppression. Ce pic correspond au passage du niveau 5 au niveau 4. En règle générale, les pics sont très prononcés. Cela est dû au fait que suite à un changement de niveau, le DST est parfaitement équilibré.

Coût moyen d'une suppression.

Nous étudions maintenant le coût moyen d'une suppression en fonction du niveau dans lequel nous nous trouvons. Pour ce faire, nous utilisons un DST avec 90 nœuds organisé en 5 niveaux. Nous supprimons 88 nœuds et nous déterminons ainsi le coût, en nombre de messages, d'une suppression à chaque niveau. Afin de ne pas fausser les résultats, nous ne tenons pas compte du nombre de messages nécessaires à la réorganisation du DST suite à la perte d'un niveau. Le tableau 1 donne le coût moyen par niveau d'une suppression dans un DST.

nb étages	1	2	3	4	5
nb messages	8	27,75	51,35	75,76	105,04

TAB. 1 – Coût moyen des suppressions dans un DST

Nous observons une croissance linéaire du nombre de messages en fonction du niveau. Nous obtenons un coût moyen d'une suppression de 53,58 et un écart type de 97,04. Dans le cas étudié, nous avons volontairement initialisé un DST fragile, puisque ses bornes sont 2 et 4. Ainsi, en étudiant le pire cas, nous obtenons la croissance la plus défavorable.
Dans la suite nous présentons les performances des algorithmes de recherche en comparant le DST à d'autres topologies.

4. Algorithmes de parcours

Les simulations de recherche que nous présentons dans la partie 6 sont basées sur deux algorithmes de parcours. Le premier, basé sur le flooding, est utilisé pour les topologies de graphes et d'arbres. Le second repose sur le DST pour réaliser un parcours avec des propriétés similaires.

4.1. Graphes et des arbres

L'algorithme utilisé [5] est identique pour les deux topologies. Il marque les nœuds qu'il traverse pour mémoriser leur état [*non-contacté* | *contacté* | *inondé*] au regard des étapes d'exécution. Un nœud *non-contacté* n'a pas encore reçu la requête. Lorsqu'il la reçoit, il devient *contacté* et lorsqu'il n'a plus de voisins *non-contacté*, il est *inondé*. Les nœuds enregistrent également l'état de leurs voisins face à cette requête. Ainsi, un nœud peut être considéré comme *contacté* par l'un des ses voisins et *non-contacté* par un autre. Initialement le nœud initiateur et ses voisins sont marqués comme *non-contactés*.
Si un nœud ne possède pas localement une ressource, il crée un identificateur de requête et envoie une requête à l'ensemble de ses voisins qui sont marqués comme *contactés*. En réponse, les voisins donnent la liste des ressources qui correspondent à la requête. Si aucune ressource n'est trouvée, l'initiateur répète

l'algorithme jusqu'à ce qu'il obtienne une liste non-vide, que la valeur de TTL soit atteinte ou que le graphe ait été entièrement parcouru.
Quand un nœud reçoit une requête pour la première fois, il retourne la liste des ressources disponibles correspondant à cette requête, positionne son état à *contacté* et marque l'ensemble de ses voisins, sauf celui dont il a reçu la requête, comme *non-contactés*. Quand le nœud reçoit une requête qu'il a déjà reçu, il fait suivre cette requête à ses voisins, qui ne sont pas marqués *inondés*, et positionne leur état à *contacté*. Pour les graphes, lorsqu'un nœud reçoit une requête d'un de ses voisins alors qu'il l'a déjà reçue d'un autre, et qu'il n'a plus de voisin à contacter ou que le TTL est atteint, il répond qu'il est inondé à l'émetteur qui modifie son état en conséquence. Dans ce cas, le voisin ne lui fera plus suivre cette requête. Dans le cas des arbres, un seul lien permet d'aboutir à un nœud, il est donc suffisant que les feuilles, ou les nœuds ayant atteint le TTL, retournent le message d'inondation. Lorsque l'initiateur a reçu une réponse d'inondation de chacun de ses voisins, le parcours de l'arbre ou du graphe est terminé.

4.2. DST

L'algorithme utilisé avec le DST est plus simple dans la mesure où il peut tirer partie de la structuration des nœuds et de leur table de routage pour ne pas avoir à identifier les requêtes ou à marquer les nœuds avec un état. L'algorithme respecte cependant les propriétés des algorithmes de flooding en contactant, de la même manière, un nombre exponentiel de nœuds à chaque itération.

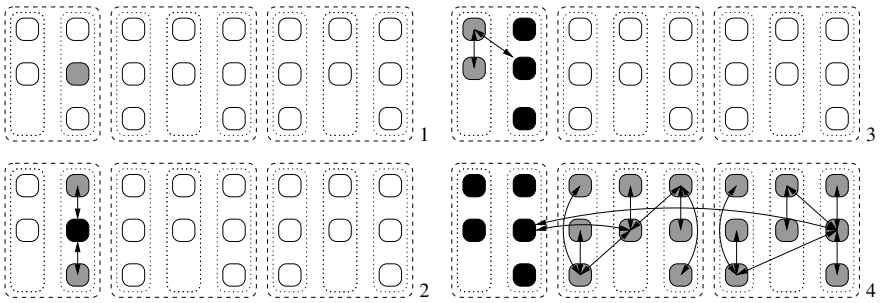

FIG. 8 – Exemple de parcours du DST

Pour lancer une requête de recherche, l'initiateur crée un compteur *depth*, positionné à 1, puis il diffuse le message de recherche dans le niveau *depth* du DST. Si cette diffusion ne permet pas de trouver la ressource, la valeur de *depth* est incrémentée et la requête est diffusée à ce niveau. Ainsi, l'algorithme s'arrête bien quand la ressource est trouvée, quand un TTL est atteint (*depth* == TTL), ou lorsque l'ensemble du DST est parcouru (*depth* est égal à la hauteur du DST). À la réception d'une requête le nœud retourne directement sa liste de ressources correspondantes s'il est une feuille du DST. Sinon il transmet la requête à ses fils, attend leur réponse à laquelle il ajoute ses propres ressources avant de répondre à son père.
La figure 8 illustre les quatre étapes de l'exécution de cet algorithme pour un DST de hauteur 3. La requête est initialisée par le nœud gris à l'étape 1. Ensuite, elle est diffusée au niveau 1 à l'étape 2, au niveau 2 à l'étape 3 et au niveau 3 à l'étape 4.

5. Description de la simulation

À l'origine, le DST devait être étudié en utilisant un modèle de réseau simulant Internet. Cependant, la complexité du modèle Internet mais aussi son hétérogénéité rend les expériences non reproductibles et les conclusions hasardeuses. Nous avons donc choisi un modèle où tous les ordinateurs possèdent un lien. Chaque lien possède une bande passante limitée, implantée sous la forme de files d'attentes FIFO, et est connecté aux autres à travers un réseau virtuel. La bande passante est, de ce fait, fixe pour un nœud. Un message a besoin d'1 ms pour traverser un lien. Ce modèle simple permet de maîtriser l'aspect communication tout en restant proche de la réalité. Vous pouvez trouver les détails complets de la simulation en téléchargeant le simulateur[3].
Nous exécutons des simulations pour des populations de 10, 100, 1 000 et 10 000 nœuds afin d'étudier le comportement des algorithmes lors d'importants changements d'échelle. Pour toutes topologies le TTL est

[3] `http://lifc.univ-fcomte.fr/~philippe/pub/simulator-12-06.tar.gz`

fixé à 10. Il y a cent types de ressources différents et chaque ordinateur a une probabilité de 10 % de posséder une ressource de chaque type. Chaque requête de recherche s'arrête, soit si elle a trouvé un ordinateur possédant la ressource, soit si le TTL est atteint ou enfin si toute la structure a été traversée. Notons que dans les simulations nous remarquons qu'une profondeur maximale de 4 est statistiquement suffisante pour trouver une ressource.
À propos des caractéristiques des topologies, les arbres sont bi-directionnels et d'arité 5. Les graphes sont aussi bi-directionnels, connexes et le degré de chaque nœud est de 5. Le DST est construit de sorte que chaque nœud possède 5 fils. Ces degrés ont été choisis parce qu'ils donnent les meilleurs performances dans nos simulations. Nous avons effectué plusieurs tests à des échelles différentes pour trouver les degrés optimaux. Nous les avons ensuite utilisés pour toutes nos simulations en considérant qu'ils sont toujours optimaux. Cependant ces valeurs dépendent de la charge des liens et de la probabilité de trouver un service. Le degré choisi n'est donc plus forcément optimal si l'un de ces paramètres est modifié.

6. Résultats des simulations

Dans ce paragraphe nous discutons des résultats obtenus lors des simulations. Nous voyons d'abord l'influence de la probabilité de
trouver un service sur les nœuds de la plate-forme de tests. Nous comparons ensuite les performances obtenues pour chacune des trois topologies testées. Les deux critères de performances pertinents ici sont le temps moyen pris par la recherche et son évolution en fonction de la charge moyenne du système. La charge du système est définie par la fréquence des requêtes lancées sur l'ensemble du système. En fin de partie, nous expliquons comment la charge se distribue sur les nœuds de la plate-forme en fonction de la topologie.

6.1. Probabilité de présence du service

La probabilité de trouver un service sur chaque nœud a une grande influence sur les performances de la recherche. La figure 9 montre, pour un DST de 10 nœuds, le temps moyen de recherche d'un service en fonction de la fréquence des requêtes pour les probabilités de présence du service recherché sur chaque nœud de 20 %, 10 %, 5 %, 2.5 % et 1.25 %.

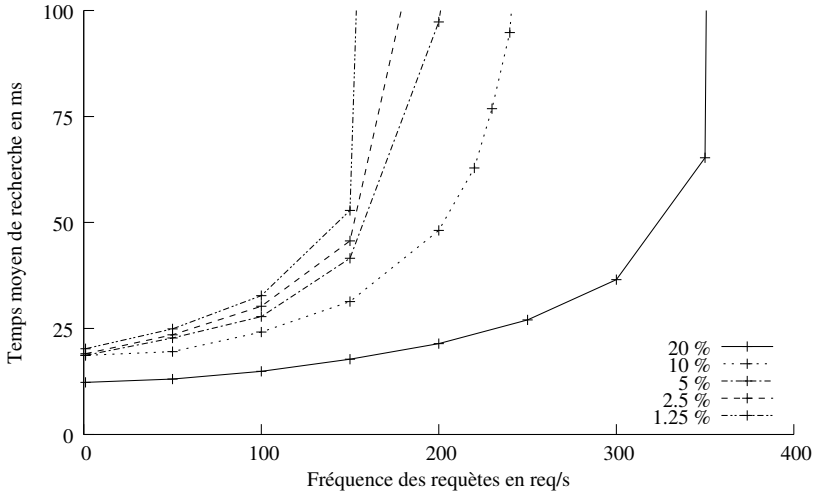

FIG. 9 – Probabilités de présence du service.

Le premier enseignement de cette expérience est qu'il est d'autant plus rapide de trouver un service que la probabilité de présence de ce service est grande sur chaque nœud. Si la probabilité de trouver le service est grande, en moyenne, peu de requêtes doivent être lancées avant de trouver le service et si nous devons contacter moins de nœuds, le temps de la recherche est réduit. Ainsi, moins de messages sont nécessaires pour mener à bien chaque recherche permettant à plus de requêtes d'être traitées par le système.
Inversement, en réduisant la probabilité de présence du service sur chaque nœud, le temps de moyen des recherches augmente et le nombre de requêtes pouvant être supportées par le système se réduit. Ce résultat est indépendant de la topologie et du nombre de nœuds.

6.2. Topologie de 10 nœuds

Les résultats des simulations menées sur une topologie de 10 nœuds sont reproduits à la figure 10. Les simulations montrent que le temps moyen nécessaire à la résolution d'une recherche dépend du nombre de requêtes. Lorsque le nombre de requêtes augmente, le système devient de plus en plus chargé et les messages prennent de plus en plus de temps avant d'être délivrés.

Lorsque le nombre de requêtes entrantes dépasse le nombre de requêtes résolues, le système sature. Cette saturation est facilement identifiable sur la figure 10 : pour le DST, le temps moyen nécessaire à la résolution d'une requête augmente d'abord lentement en fonction de la fréquence, puis croît très fortement à partir de 200 req.s^{-1}. Par contre, les topologies en forme de graphe ou d'arbre de 10 nœuds saturent beaucoup plus rapidement.

Certaines requêtes traversent le graphe dans sa totalité lorsque le service n'est pas trouvé (probabilité de présence du service de 10% sur 10 nœuds). Par conséquent, ces requêtes demandent une autre étape pour vérifier qu'aucun autre nœud ne peut être contacté. Lors de cette dernière étape, les messages sont envoyés sur les derniers liens non encore parcourus, dans les deux sens, et en direction de nœuds déjà traversés au moins une fois. À cause de cette dernière étape, le coût de la traversée des nœuds dépend du nombre de lien plutôt que du nombre de nœuds et affecte le temps moyen de recherche.

Le DST a le meilleur comportement pour ces simulations. Comme un DST est basé sur des arbres couvrants (un par nœud), chaque départ d'une requête de recherche part d'un nœud, racine de son arbre couvrant, et consomme au plus $2.n$ messages avec n nœuds. Donc le DST ne connaît pas le goulet d'étranglement habituel des arbres et peut supporter une charge plus importante que d'autres topologies avec 10 nœuds. Il faut également noter que le DST génère un temps de recherche inférieur aux arbres lorsque le système n'est pas chargé (voir le paragraphe 6.3).

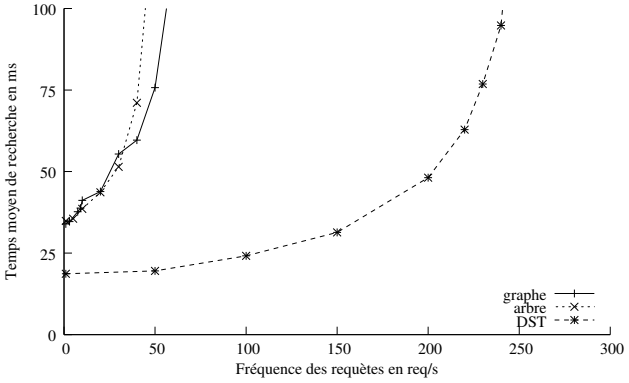

FIG. 10 – Performances pour 10 nœuds

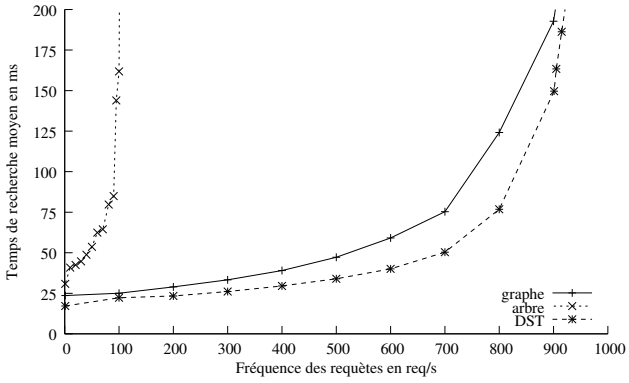

FIG. 11 – Performances pour 100 nœuds

6.3. Topologies de 100 nœuds

La figure 11 présente les résultats obtenus pour des simulations sur 100 nœuds. Comme précédemment, les trois topologies saturent dès que l'on dépasse une certaine fréquence d'arrivée de requêtes. Ici encore, les arbres supportent très mal l'augmentation de la charge. Avec 100 req.s^{-1} les arbres sont surchargés alors que les graphes et le DST peuvent supporter jusqu'à 700 req.s^{-1}. Avant d'arriver à ce niveau, le temps moyen de recherche du service augmente lentement avec la charge moyenne du système.

Les performances obtenues avec les graphes et le DST sont similaires en termes de résistance à la charge et de temps moyen de recherches de services. Pour comprendre cette similitude il faut se rappeler que le service recherché a une probabilité de présence de 10 % sur chaque nœud, que la recherche s'arrête dès qu'un service est trouvé et que les graphes ne sont pas *petit monde*. Pour les graphes de taille 100 et plus, les recherches aboutissent favorablement en une ou deux étapes, trois étapes étant un cas exceptionnel et les cherches demandant plus de trois étapes n'existent pas. Une recherche en deux coups signifie qu'un maximum de 31 nœuds[4] sont parcourus, d'où une probabilité pour un nœud d'être contacté au moins deux fois inférieure à $\frac{1}{3}$. Si chaque nœud d'un graphe est contacté seulement une fois, alors seulement $2.n$ messages sont nécessaires pour atteindre les n nœuds de la plate-forme. De plus, ce nombre moyen de

[4] $31 = 5^0 + 5^1 + 5^2$.

messages est optimal. Étant donné que chaque recherche nécessite d'atteindre une petite partie du graphe, peu de liens sont utilisés pour accéder à des nœuds déjà contactés. Donc, peu de nœuds sont contactés deux fois pour une même recherche et le nombre d'échanges de messages est, en première approximation, à peu près similaire à ce que l'on trouve avec les arbres couvrants. Ceci explique les similitudes évoquées plus haut.

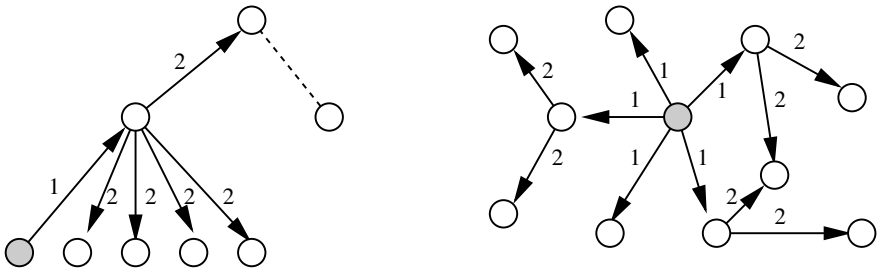

FIG. 12 – Les deux premières étapes de la recherche dans un arbre et dans un graphe

D'autre part, nous observons que les arbres ont un temps moyen de recherche plus grand que les graphes même avec une charge moyenne faible. Cela est surprenant si nous pensons que le temps moyen d'une recherche dépend seulement du nombre de messages échangés et qu'un parcours complet d'un arbre requière un nombre optimal de messages. Les raisons de ce comportement sont illustrées à la figure 12. Dans un arbre équilibré, il y a toujours plus de feuilles que de nœuds internes. Comme chaque nœud a la même probabilité d'initier une recherche, la majorité des recherches sont initiées par les feuilles de l'arbre. Ainsi, avec des topologies d'arité 5, le nombre de nouveaux nœuds rencontrés lors de la recherche partant d'une feuille est 1 à la première étape, 5 à la deuxième, puis encore 5, puis 25, 25, etc. Pour le DST, ce nombre suit les puissances de 5 alors que dans le cas des graphes, le nombre de nouveaux nœuds contactés à chaque étape se situe entre les valeurs trouvées pour les arbres et le DST, plutôt proche des valeurs du DST étant donnés les résultats des simulations. Ceci explique les mauvaises performances observées pour la topologie en arbre dans ce cas.

6.4. De 1 000 et 10 000 nœuds

Les figures 13 et 14 rendent compte des simulations effectuées respectivement sur des plates-formes de 1 000 et 10 000 de nœuds. Les arbres de taille 1 000 et 10 000 sont saturés pour des charges respectivement de 300 req.s^{-1} et 1 000 req.s^{-1}. En revanche, la surcharge n'est visible sur les graphes et le DST qu'à partir de 8 000 req.s^{-1} et 40 000 req.s^{-1} pour des tailles respectives de 1 000 et 10 000 nœuds. Avant d'être surchargés, le temps moyen de recherche dans un DST ou dans un graphe croît lentement avec l'augmentation de la fréquence des requêtes.

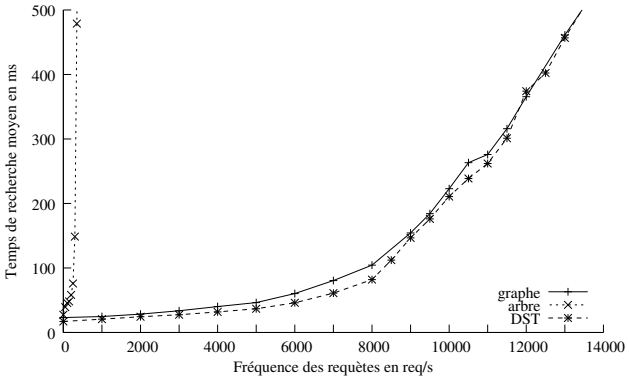

FIG. 13 – Performances pour 1 000 nœuds

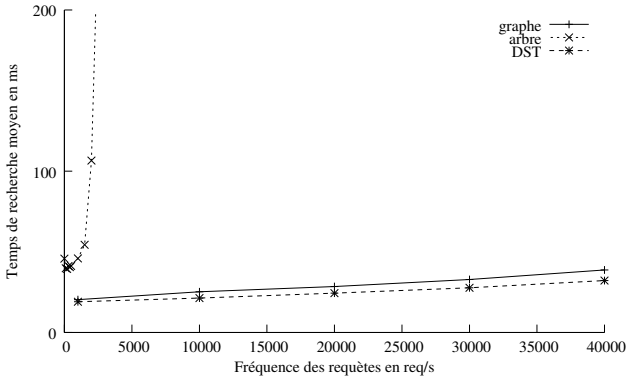

FIG. 14 – Performances pour 10 000 nœuds

Ici encore, les performances rencontrées avec les graphes sont comparables à celles obtenues avec le DST

pour les mêmes raisons que précédemment.
Pour des raisons liées aux capacités mémoires des machines utilisées pour ces simulations, il n'a pas été possible d'augmenter la fréquence des requêtes au delà de 40 000 req.s^{-1}. Au delà de cette limite, la machine swappe et le processeur n'est plus utilisé qu'à 3 % de sa capacité initiale, ce qui allonge le temps de simulation à plus d'un mois.

6.5. Équilibrage de la charge

Nous avons affirmé précédemment que les graphes et le DST distribuaient la charge sur les nœuds alors que les arbres souffraient de goulets d'étranglement. L'étude de la charge des nœuds au cours de nos simulations corroborent cette affirmation. La figure 15 en est une illustration pour dans le cas d'un arbre de 1 000 nœuds. Chaque point de cette figure représente un nœud de la plate-forme et illustre le pourcentage des messages envoyés par ce nœud au cours de la simulation. La charge est ici très mal répartie puisque certains nœuds envoient 20 à 40 fois plus de messages que la majorité des autres. Par contre, dans le cas des graphes ou du DST, nous remarquons que la charge est assez bien répartie.

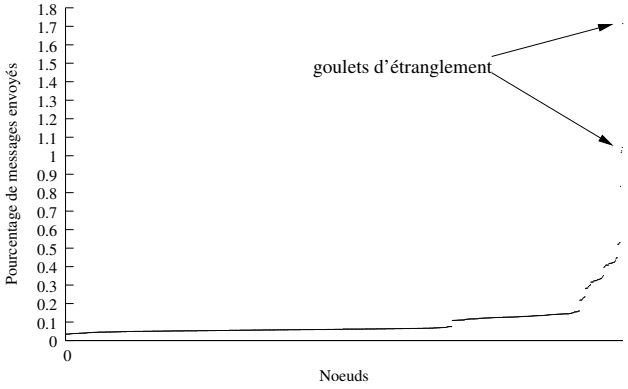

FIG. 15 – Charge dans un arbre de 1 000 nœuds

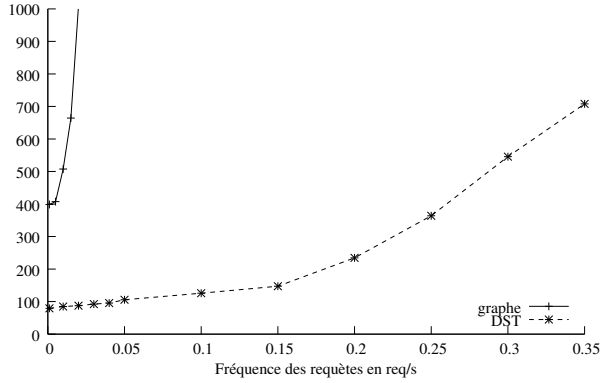

FIG. 16 – Performances pour 1 000 nœuds alors que le service recherché n'est pas disponible

6.6. Extensibilité des trois topologies

Jusqu'alors, nous avons discuté des performances observées pour chaque taille, indépendamment les unes des autres. Ici, nous présentons comment le nombre de nœuds influence les performances de chaque topologie. Des figures 10, 11, 13 et 14 nous notons qu'ajouter de nouveaux nœuds permet d'accroître les performances en terme de charge supportée. En effet, dans tous les cas, la charge supportée par nœud décroît.
Pour la topologie en arbre, des charges de 40 req.s^{-1}, 100 req.s^{-1}, 300 req.s^{-1} et 1 000 req.s^{-1} saturent respectivement des tailles d'arbres de 10, 100, 1 000 et 10 000 nœuds. Ici, multiplier la taille de l'arbre par 100 ne permet pas de supporter une augmentation de charge de plus d'un facteur 10. Nous pouvons conclure que cette topologie n'est pas extensible.
À l'inverse, le DST permet une bonne l'extensibilité. La charge supportée augmente linéairement avec le nombre de nœuds : les charges de 150 req.s^{-1}, 700 req.s^{-1}, 8 000 req.s^{-1}et plus de 40 000 req.s^{-1} saturent respectivement des DST de taille 10, 100, 1 000 et 10 000 nœuds. Hors saturation, les performances en terme de temps de recherche moyen restent stables : ce temps varie de 25 ms à 75 ms quelle que soit la taille du DST. Un résultat trouvé pour 150 req.s^{-1} sur 10 nœuds d'un DST peut paraître inconsistant avec la performance de 700 req.s^{-1} pour 100 nœuds, mais cela est normal. Comme il n'y a que 10 nœuds, une recherche en deux étapes contacte les 10 nœuds alors que la même recherche contacte 25 nœuds avec un DST de taille 100. Moins de nœuds contactés signifie moins de messages envoyés et implique que la plate-forme supporte plus de charge.
L'extensibilité du graphe est similaire à celle du DST sauf pour les petites échelles où cela est faux. Cette ressemblance tient au fait que ces deux topologies se comporte de la même manière lorsque la probabilité de trouver un service sur chaque nœud est fixée (10 % dans le cas des simulations) et que le nombre de nœuds assez grand. Par contre, si l'échelle est trop petite, alors les requêtes contactent plusieurs fois les mêmes nœuds et le graphes perdent en performances.

Cependant, si la probabilité de trouver un service est assez basse pour que les nœuds reçoivent la demande plusieurs fois, alors les performances observées avec les graphes se dégradent par rapport à celles obtenues avec le DST. Ainsi La figure 16 présente les performances observées sur un graphe et sur un DST de taille 1 000 dans le cas extrême où la probabilité de présence du service recherché est nulle. Ce qui signifie que la topologie doit être complètement parcourue pour que chaque nœuds puisse retourner une réponse négative. Le DST envoie alors moins de messages et distribue la charge sur l'ensemble des nœuds. Nous noterons qu'une fréquence d'une requête toutes les 5 secondes est suffisante pour saturer dans ce cas un DST de taille 1 000. Cela est normal car les algorithmes d'inondation de type TTL génèrent plusieurs vagues de recherche et ne sont pas efficaces dans ce cas. L'utilisation des DHT seraient plus indiquées ici.

7. Conclusion

Si les performances des algorithmes de recherche ne dépendaient que du nombre de messages, en théorie, les arbres seraient plus efficaces que les graphes. Seulement, il faut tenir compte des goulets d'étranglement et à ce niveau les graphes résistent mieux que les arbres.
Le Distributed Spanning Tree (DST) est une structure originale basée sur des arbres de recouvrement permettant la connexion de plusieurs ordinateurs tout en distribuant la charge des parents parmi leurs fils. Cette structure est théoriquement plus performante que les arbres et les graphes pour la recherche par inondation. Le DST a été conçu pour envoyer un nombre de messages optimal tout en évitant les goulets. La simulation présentée dans cet article confirme l'intérêt de cette approche et le DST obtient toujours de meilleurs résultats que les deux autres topologies en terme de rapidité de recherche et de support de la charge. De plus, nous avons observé que grâce à la structure du DST, l'implémentation d'autres algorithmes de parcours est plus simple que pour un graphe.
Les résultats obtenus nous incitent à continuer nos expérimentations. Il est nécessaire, avant de pouvoir affirmer que la topologie est utilisable comme overlay, de valider son comportement dans un environnement dynamique. Nous privilégions cet axe pour nos futures recherches.

Bibliographie

1. James Aspnes and Gauri Shah. Skip graphs. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, 2003.
2. E. Caron and al. A scalable approach to network enabled servers. In 8^{th} *EuroPar Conference*, volume 2400 of *LNCS*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.
3. S. Dahan. *Mécanismes de Recherche de Services Extensibles pour les Environnements de Grilles de Calcul*. PhD thesis, U.F.R. des Sciences et Techniques de l'Université de Franche-Comté, December 2005.
4. S. Dahan. Simulator and data used for the rempar'07 article, 2006. `http://lifc.univ-fcomte.fr/~dahan/dst/rempar07-sim.tar.gz`.
5. S. Dahan, J.-M. Nicod, and L. Philippe. Scalability in a GRID server discovery mechanism. In *10th IEEE Int. Workshop on Future Trends of Distributed Computing Systems, FTDCS 2004*, pages 46–51, Suzhou, China, May 2004. IEEE Press.
6. S. Dahan, J.-M. Nicod, and L. Philippe. The Distributed Spanning Tree : A scalable interconnection topology for efficient and equitable traversal. In *Proceedings of CCGrid 2005 Workshop Global and Peer-2-Peer Computing*, Cardiff, UK, May 2005. IEEE Press. CD-ROM.
7. G. Fletcher and Sheth H. Unstructured peer-to-peer networks : Topological properties and search performance. In 3^{rd} *Int Workshop on Agents and Peer-to-Peer Computing (AP2PC) at AAMAS 2004*, pages 14–27, New York, 2004.
8. R. Gaeta, G. Balbo, S. Bruell, M. Gribaudo, and M. Sereno. A simple analytical framezork to analyze search strategies in large-scale peer-to-peer networks. *Performance Evaluation*, 62(1–4) :1–16, Oct. 2005.
9. D. E. Knuth. *The Art of Computer Programming*, volume 3, chapter 6.2.4. Addison-Wesley, 75 Arlington Street, Suite 300, Boston, MA 02116, 1998.
10. E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lims. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Survey and Tutorial*, 7(2), April 2005.
11. R. E. McGrath. Discovery and its discontents : Discovery protocols for ubiquitous computing. Technical Report UIUCDCS-R-99-2132, Department of Computer Science University of Illinois, Champaign, IL, USA, April 2000.
12. L. Qin, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *ACM SIGMETRICS 2002*, pages 258–259, Marina Del Rey, California, 2002.
13. Zhang Zhenjie, Yu Feng, and Yang Xiaoyan. Report on unstructured network in peer-to-peer system, 2004.